Evaluating Contexts in OpenSHMEM-X Reference Implementation

Aurelien Bouteiller[2*], Swaroop Pophale[1*], Swen Boehm[1], Matthew B. Baker[1], and Manjunath Gorentla Venkata[1]
* Contributed Equally

[1] Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
[2] Innovative Computing Laboratory
The University of Tennessee
Knoxville, USA

Abstract. Many-core processors are now ubiquitous in supercomputing. This evolution pushes toward the adoption of mixed models in which cores are exploited with threading models (and related programming abstractions, such as OpenMP), while communication between distributed memory domains employ a communication *Application Programming Interface* (API). OpenSHMEM is a partitioned global address space communication specification that exposes one-sided and synchronization operations. As the threaded semantics of OpenSHMEM are being fleshed out by its standardization committee, it is important to assess the soundness of the proposed concepts. This paper implements and evaluate the "context" extension in relation to threaded operations. We discuss the implementation challenges of the context and the associated API in *OpenSHMEM-X*. We then evaluate its performance in threaded situations on the Infiniband network using micro-benchmarks and the Random Access benchmark and see that adding communication contexts significantly improves message rate achievable by the executing multi-threaded PEs.

Keywords: PGAS • Shared Memory • Interoperability

1 Introduction

OpenSHMEM [5] library specification aims at providing a standard API for SHMEM libraries to aid portability across different vendor implementations on different architectures. With behavior and semantics defined for the library API, OpenSHMEM programs are expected to provide consistent behavior across multiple platforms. Many high-performance implementations from vendors such as HPE, Cray, and Intel along with open source implementations by University of Houston (reference implementation over GASNet) and ORNL (reference implementation over UCX, *OpenSHMEM-X*) exist.

As we march towards the exascale era, the new trend in hardware is to have more capable nodes with multiple levels of concurrency and parallelism. To utilize these complex systems OpenSHMEM programming model has to evolve to include support for heterogeneous architectures and hybrid programming. The first step towards it is to have thread safety as part of the OpenSHMEM specification that will allow threads representing a PE to make OpenSHMEM calls.

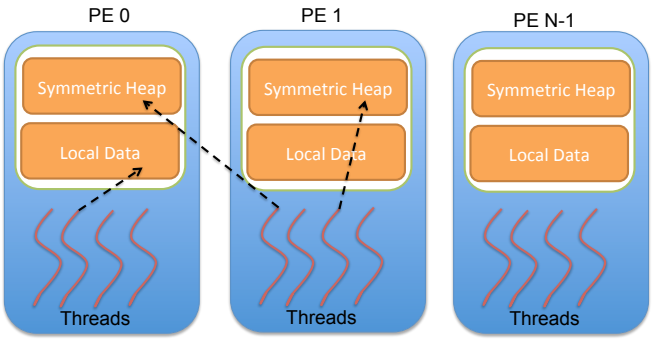

Fig. 1. Multi-threaded OpenSHMEM Memory Model

Figure 1 shows the logical evolution of the OpenSHMEM memory model, where each PE may spawn a number of threads. Since the OpenSHMEM programming model does not recognize individual threads as separate entities, any OpenSHMEM operation initiated by a thread is considered an action of the PE as a whole. Like the single-threaded PE, threads of a multi-threaded PE may access the local memory of the PE and the *symmetric* memory of its PE and that of any other PE participating in the OpenSHMEM application but cannot access the local memory of any other PE. The threading model itself is outside the scope of this work.

The most recent version, 1.3 of the OpenSHMEM specification, does not define a thread safe API, but the community is having a productive discussion about the appropriate interfaces and semantics that can effectively support concurrently communicating threads. Multiple designs have been proposed in recent

years, and in this paper we evaluate the *Contexts* proposal by providing an implementation over IB using the UCX communication library. The paper also explores the possibility of a multi-threaded PE where individual threads can, within the bounds of the OpenSHMEM semantics, add another level of concurrency to the application thus increasing overlap not only between operations of the different PEs but also that of the same PE.

In Section 2, we give the background on thread safety in OpenSHMEM and details about the *Context* proposal as presented to the OpenSHMEM community. In Section 3, we discuss our implementation and the different considerations made to make the implementation thread safe. In Section 4, we discuss the different benchmarks used to evaluate the implementation and discuss our results. Related work in this context is covered in Section 5. The highlights of our analysis and our next steps are discussed in Section 6.

2 Background

Accompanying the evolution toward many-core architectures, hybrid programming models are becoming increasingly popular and multiple threads play an ever increasing role in improving the utilization and programmability of modern High-Performance Computing (HPC) systems.

The OpenSHMEM Specification does not currently have a threading model, thus hybrid programming (e.g. using OpenMP or OpenACC in OpenSHMEM programs) is not directly supported. Nonetheless, successful use of OpenACC has been demonstrated [2], with the limitation that OpenACC calls are restrained to isolated segments in the OpenSHMEM program between *shmem_barrier_all* calls. In contrast to this work, we propose to study the support of threaded Processing Elements (PEs) where individual threads may participate in OpenSHMEM calls on behalf of a PE. Through the context proposal, hardware resources are exposed to these threads, which in turn provide concurrency and the ability to synchronize on a per thread basis.

2.1 Thread safe API

An earlier proposal from Cray [4] introduced OpenSHMEM extensions for thread safety which include different levels of threading support similar to those used in MPI. This proposal has not been incorporated by the OpenSHMEM specification community as of version 1.3, but the OpenSHMEM standardization committee is working on integrating these concepts in the standard.

These thread support levels are hierarchical. In lower support levels, the programmer is responsible for restraining access to authorized threads, or serializing access to the OpenSHMEM library, according to the supported semantics in that level. Higher levels expand on the capabilities of the lower levels by relaxing the allowed concurrency between OpenSHMEM calls. In the highest thread support level, multithreaded programs may issue concurrent calls to the otherwise unchanged OpenSHMEM routines.

The first level of thread support is `SHMEM_THREAD_SINGLE` and it allows for one thread per process. The next level `SHMEM_THREAD_FUNNELED` , permits processes to have multiple threads but only one of the threads can make OpenSHMEM calls, and it is the programmer's responsibility to enforce that all OpenSHMEM calls in a process are executed by that thread. The next level of thread support is `SHMEM_THREAD_SERIALIZED`, which permits multiple threads to issue OpenSHMEM calls in a PE, but only one OpenSHMEM call per process can be active at any given time. Since simultaneous calls from two threads belonging to the same process are not allowed, it limits concurrency available through the multithreading approach. Finally, in the `SHMEM_THREAD_MULTIPLE` level, processes may have multiple threads and any thread may issue a OpenSHMEM call at any time, subject to a few restrictions. In this latest mode, the OpenSHMEM Specification would enable hybrid programming with OpenSHMEM, but it would still overlook other performance important factors such as network resource management. Modern HPC networks have sophisticated network adapters, often exposing multiple resources to the software stack. To utilize these resources in a optimal way, the application programmer needs to be able to provide a "hint" to the communication library about the intended, and potentially exclusive, usage of the OpenSHMEM interfaces, and the OpenSHMEM interface needs to be expanded to expose the available resources in a portable way to the programmer.

2.2 Communication Contexts

The context proposal, first fleshed out by Dinan*et. al.* [7], seeks to alleviate the aforementioned issue with resource allocation, mapping, and sharing. The context proposal which expands on top of the thread safe API discussed in Section 2.1, introduces a set of new API functions to map contexts onto resources, and to issue Remote Memory Access (RMA) operations and synchronizations on a specific context.

Additional RMA operations from the context proposal take the context as an explicit parameter. They follow OpenSHMEMs API for the function names and add a `ctx_` qualifier to the function name (*e.g.,* `shmem_ctx_putmem` corresponds to `shmem_putmem`). Finally, there are supplementary API functions for memory ordering using contexts (`shmem_ctx_fence` and `shmem_ctx_quiet`). The full semantics of the proposal are outside the scope of the paper; the interested reader will find details on the proposed API in [7].

Contexts are created using the `shmem_ctx_create` function. The user may indicate at creation time if the context is intended for shared, or exclusive use. When the context is shared, multiple threads may issue operations concurrently on the context. In addition to context synchronization targeting explicitly the context, non-context synchronizations (*e.g.,* `shmem_quiet`, which does not have a context argument) will synchronize the default context (*i.e.,* the implicit context in which operations without a context argument operate). When the context is exclusive, only the calling thread may issue operations on the context, and only synchronization operations that explicitly target the context affect the ordering, visibility and completion of posted communication on the context.

Thanks to these additional API calls, the context proposal enables several advantageous usage patterns. Regardless of the number of threads in an application, contexts provide the opportunity to finely control and pipeline overlap of multiple communication. Contexts can in effect be used to provide a fine control on completion ordering while maintaining the performance benefit of implicit operations (*i.e.,* sparring the overhead associated with request-based approaches).

Additionally, RMA operations in one thread can be issued on a separate context, and thereby isolated from RMA operations issued by a different thread on another context. As we will further discuss in the rest of this paper, supporting thread safe access to shared network queues and resources can have a significant impact on communication performance, most notably on latency and injection rate as locks, atomic operations and memory barriers must be added to the critical path to ensure consistency across multiple threads. If a context is exclusive, although the application may issue operations from multiple threads, there is no need to lock or ensure memory consistency between these threads, therefore opening an alley for lock-free multithreaded operations.

Last, given the information about the number and nature (private or shared) of the contexts requested by the application threads, and the number of underlying hardware communication channels, the OpenSHMEM implementation has an opportunity for optimizing the mapping between contexts and resources to again minimize the need for serialization and locking. If sufficient hardware resources are available, locking can be eliminated completely.

3 Implementation

Early evaluation versions of the context proposal have been implemented targeting specific hardware such as Portals, and Cray DMAPP [14]. In this work, we implement the concepts in a generic fashion in the OpenSHMEM reference implementation, with the intent of stressing the portability implications of the proposal when facing a generic interface to access network resources.

To test the proposed extensions to OpenSHMEM, we implemented the context proposal in the OpenSHMEM reference implementation. The reference implementation can use different conduits to map OpenSHMEM calls onto network operations. The University of Houston (UH) reference implementation employs GASNet [3], but the OpenSHMEM reference implementation has recently been adapted to use UCX as the networking conduit [1].

UCX is a community effort to implement a portable and scalable high performance network API framework. UCX supports different HPC networking architectures and supports RMA operations, active messages, as well as tag matching. UCX provides different sets of APIs. UCP is the high level API,and is used by our *SHMEM* implementation. Additionally, UCX provides the UCT API, which abstracts, at a low level, the differences of various hardware architectures. Finally, the service API, UCS, provides functionality to write a portable networking framework. One of the strong design points in UCX is its emphasis in providing

a close to the metal, yet portable access to network technology. In addition, a significant effort has been made to make UCX API thread-scalable [11].

The UCP layer provides communication resources called UCP workers. A worker object has methods for synchronizing and ordering (similar to those provided by `shmem_quiet` and `shmem_fence`), as well as methods for waiting on events or memory locations (similar to `shmem_wait_int`). Multiple workers can be created in a process, thereby providing separated access to low level communication resources. Communication operations are posted in an UCP endpoint; An endpoint is a local queue for posting operations into a peer to peer channel between a local and a remote worker. Workers at a target can be individually addressed by creating different endpoints.

3.1 Per-context Endpoints

The major difference between the context-extended OpenSHMEM and the regular UCX OpenSHMEM is the way UCP endpoints are managed. In the regular OpenSHMEM, one UCP endpoint is created between two PEs. That endpoint is used to issue all communication and synchronization targeting that PE. With this design, when multiple thread issue OpenSHMEM calls simultaneously, all operations are serialized in that unique UCP endpoint; and the cost of thread consistency (*i.e.,* atomic operations and memory barriers) must be paid on performance critical operations (like `shmem_int_put`). In addition, a single UCP progress entity (a UCP Worker) is created. All threads requiring progress, ordering, and synchronization then issue a call to the same Worker object, which again needs to ensure internal thread safety, and enforces ordering between events that are not in a deterministic order. For example, a fence issued by one thread may enforce the order between messages issued in another thread, but without further user-level thread synchronization, that order is nondeterministic, and enforcing it is spurious overhead.

In contrast, the context extended implementation creates a supplementary worker for each calling thread. Every time a thread creates a context, a thread-specific UCP worker is instantiated on which context synchronization operations are posted. In addition, supplementary thread-specific endpoints are created to issue communication operations. All context communication calls (*e.g.,* `shmem_ctx_int_put`) are then simply remapped to the corresponding UCX call on the context-specific UCP endpoint. Similarly, synchronization calls are remapped to the context-specific UCP worker.

The context object contains a reference to the associated worker and array of context-specific endpoints. Because the context is then passed as an explicit parameter to all communication and synchronization functions, retrieving the thread-appropriate worker and endpoints can be implemented without relying on perhaps expensive thread-local storage (TLS).

3.2 Thread/Context Mapping

As the context committee is still fleshing out the context proposal, parts of the interface are still under development. One such aspect of the proposal that remains a work in progress is the mapping interface to attach contexts to threads.

The context proposal itself does not necessarily binds contexts and threads. A single thread may create and use multiple contexts, or a context can be created shareable, in which case it may be used by multiple threads concurrently, or transitioned from thread to thread for serial accesses. However, one of the most promising aspect in terms of performance optimization is for thread-exclusive contexts, that is, contexts created for the exclusive use of the thread calling the context creation function.

Our implementation of the proposal uses OpenMP runtime functions to gather information of the number of threads the application is using. Shared contexts are mapped onto the default UCX level network resource, *i.e.,* the thread-safe worker that handles operations posted on the default context, or using the non-context interfaces. For thread-exclusive contexts, UCX level network resources are allocated on a per thread basis. If the current thread has no resources assigned to it, when that thread first creates a context, a new UCX worker is created, and the reference to the worker is stored in both the new context, and in a global table indexed per thread. Note however that during communication operations, the appropriate worker resource is obtained from the context itself, unlike in Cray's thread hot implementation, thereby sparing the potentially costly identification of the calling thread and the global lookup in the shared table.

With this policy, if a thread creates multiple contexts, they will share the same resource object. This mapping policy may result in under-utilizing available hardware network resources, if the number of available network hardware queues is larger than the number of threads in the application. It also results in over-synchronizing operations issued on multiple context from the same thread, thereby limiting pipelining opportunities. An alternate strategy would be to allocate a new UCX worker for every created context, resulting in a workers potentially outnumbering the available network hardware queues. In this case, without additional information on the intended usage pattern for the context, determining which context should share a worker, so that the total number of created workers is commensurate with the available hardware resources becomes a difficult problem. A particularly stringent problem arises when one threads creates many contexts, thereby exhausting exclusive resources and leaving all other threads to suffer from the performance penalty of using shared resources. A companion proposal to the context interface adds the concept of *domains*. Domains represent a way for application codes to express preferences for the bundling of contexts on the same physical network resources, when sharing becomes necessary. In the current work, contexts are implicitly bundled by calling threads, as identified by calling OpenMP routines. Using the explicit domain interface would remove the runtime dependency to OpenMP and support arbitrary thread libraries (including full-user threads that would not register in

system calls). The domain proposal is currently evaluating exposing to the end-user the number of available hardware queues, or on the opposite, obtain from the user the desired number of logical queues and their sharing dependencies, which would then inform the communication driver mapping (such a feature is available in UCX).

4 Experimentation and Results

The experiments were conducted on a 16 node cluster at Oak Ridge National Laboratory (ORNL). Each node has two *Intel*® *Xeon*® E5-2660 processors with 10 physical cores and hyper threading, a Mellanox ConnectX-4 VPI adapter card, EDR IB (100Gb/s) and 12GB RAM.

To evaluate the implementation of *Contexts* in OSH-X we use three benchmarks. These benchmarks create a communication context per thread and all RMA and atomic operations use an additional parameter to identify the context used as discussed in Section 2.2. This isolates the different threads form each other and enables efficient resource sharing between threads. These benchmarks are:

1. Message Rate benchmark
   We modified the Ohio State University (OSU) [13] OpenSHMEM message rate benchmark to use communication contexts and OpenMP. Adding support for OpenMP enabled the benchmark to run in a threaded environment. The benchmark measures the message rate using put operations with an increasing number of PEs per compute node.
2. Bandwidth benchmark
   This benchmark has been derived by adding per thread bandwidth measurement to OSU [13] OpenSHMEM bandwidth benchmark. We measure the aggregate bandwidth across all threads of a PE, as well as the average bandwidth for individual threads across runs. The benchmark is run with two PEs on two nodes and records bandwidth of put operations with an increasing number of OpenMP threads per PE.
3. Random Access benchmark
   This benchmarks gives the Giga Updates per second (GUPs) [8] for a given number of PEs. The problem size (number of updates) is directly proportional to the number of PEs. The OpenSHMEM random access benchmark is modified such that all updates are made to remote PEs. We run different problem sizes with increasing number of OpenMP threads per PE with one PE per node.

4.1 Message Rates

We measure the message rate using `shmem_put` operations. Figure 2 shows the message rate with an increasing number of threads for PEs using communication contexts for a one byte transfer size. The overall message rate increases but the

aggregated message rate does not level off with the maximum number of threads, showing potential for an increasing number of threads to achieve higher message rate. Figure 3 shows the message rate change with increasing message size for different threads per PE.

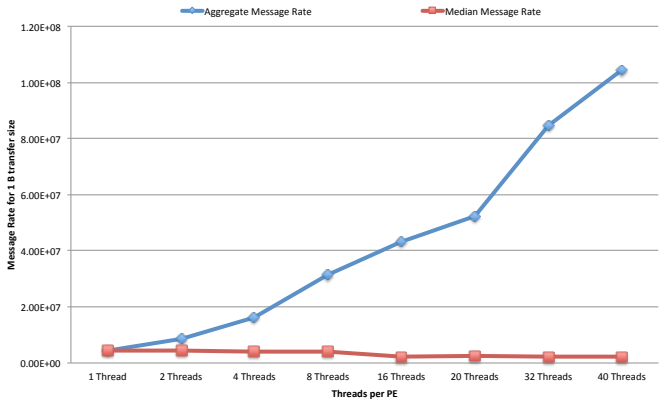

Fig. 2. Aggregate and median message rate with increasing threads per PE

4.2 Bandwidth

In Figure 5, we show the median bandwidth achieved across the thread team and the aggregate bandwidth for increasing number of threads per PE for large messages (1MB). From the graph it is evident that it is not difficult to saturate the network using large message sizes with fewer threads per PE. Figure 6 shows the bandwidth utilization as a function of transfer size. The graph shows that the benefits of having a multiple threads with independent contexts dwindle as the transfer size increases. At this point the network becomes the bottleneck.

4.3 Random Access

The OpenSHMEM Random Access benchmark allocates a large remotely accessible table data structure on each PE and updates are made to random locations at each iteration. We test this benchmark by allocating one PE per node and then increasing the number of OpenMP threads per PE. Figure 7 shows that, higher thread counts achieve higher GUPs for the same number of PEs (hence the same problem size).

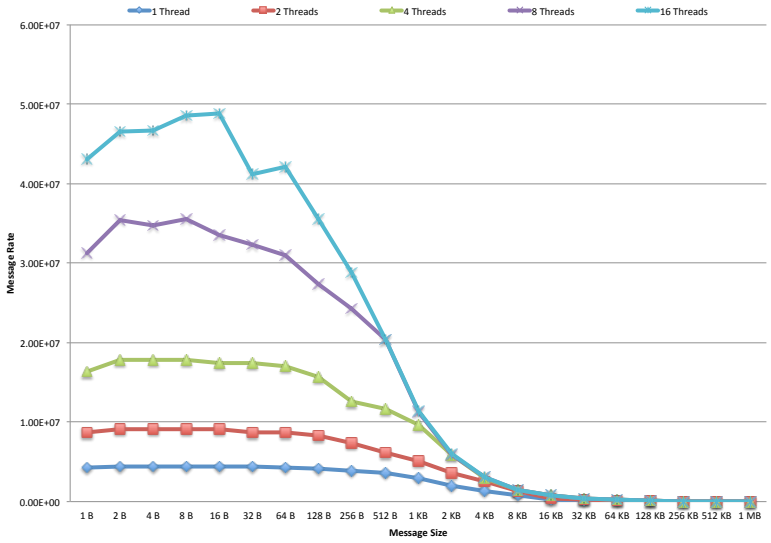

Fig. 3. Aggregate message rate (messages per second) with increasing message size for increasing threads per PE

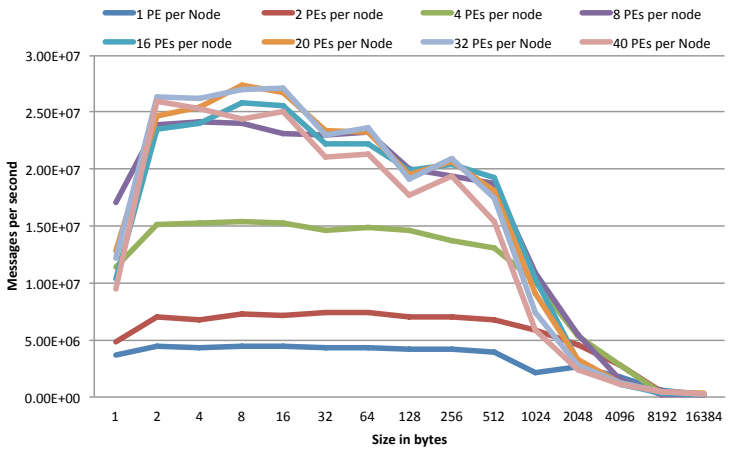

Fig. 4. Message rate as a function of the message size

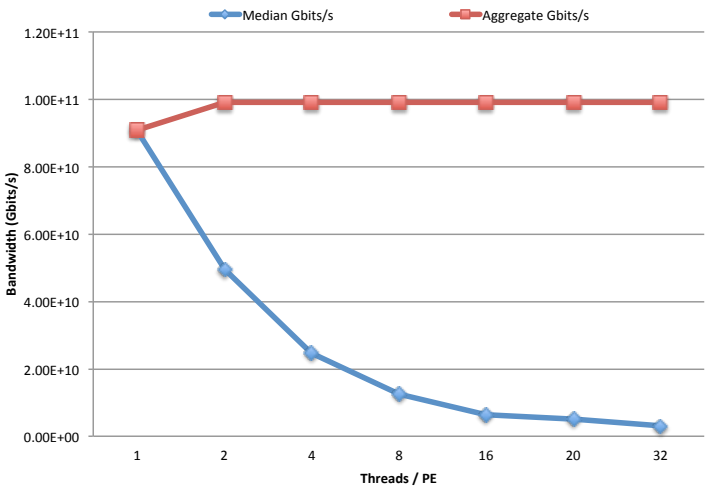

Fig. 5. Bandwidth for the context implementation for a transfer size of 1MB with increasing number of threads per PE

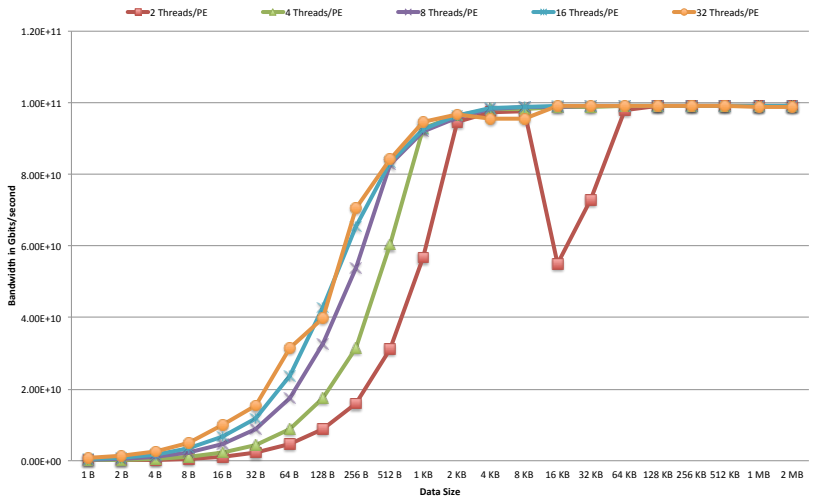

Fig. 6. Bandwidth of different threads per PE as a function of the message size

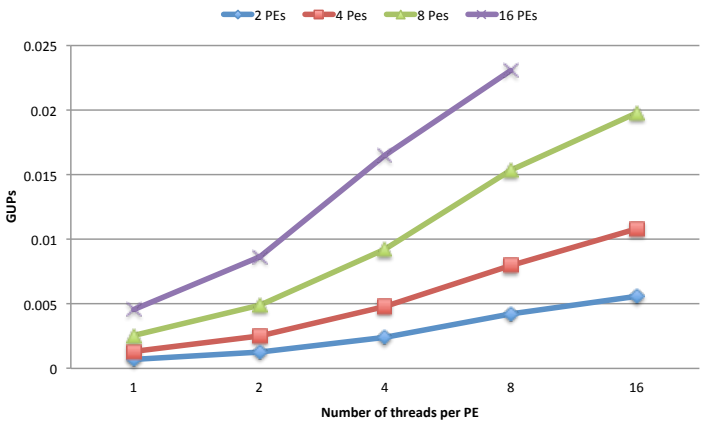

Fig. 7. GUPs measurement for different PE and threads/PE combination

5 Related Work

The OpenSHMEM communication context proposal was introduced by [7] with a prototype implementation on Portals 4 network programming interface. The proposal was later developed to include Domains; which are a group of contexts that have the same properties. Another evaluation for Cray was explored in [9] where they implemented the context and domain concepts over the Cray DMAPP (Distributed Shared Memory Application) library.

The MPI community has explored *endpoints* [6, 12] for MPICH for hybrid programming support. The difference in the two proposals is that they generated additional MPI ranks that could be assigned to threads used in the execution of such models. The MPI *endpoints* are similar to context as they also identify independent set of resources that will support concurrent communication operations. Other Partitioned Global Address Space (PGAS) low level libraries like GASNet [3] and ARMCI [10] provide ways to aggregate non-blocking communication. These aggregations can then, by the library, provide independent resources without user intervention.

6 Conclusions and Future Work

We have implemented the context proposal with a hardware neutral interposition layer (UCX) in the reference OpenSHMEM implementation. This implementation creates separate resource pools on a per-thread basis, thereby eliminating the cost of serializing accesses from multiple threads to a single shared resource. Our experimental evaluation demonstrates that the design can scale with the number of threads concurrently issuing communication. Notably, the random access benchmark performs significantly faster with more threads per PE.

In future works, given the significant advantage shown in the micro-benchmarks and Random Access benchmark, we expect the performance gain to permeate to the applications; we would therefore like to extend the evaluation to more application benchmarks. As we stated in the implementation section, the mapping of contexts to hardware resources remains an open problem at this point. Given our experience with UCX, we would like to experiment with the concept of domains to drive the context mapping onto hardware resources in an efficient and portable way.

Acknowledgment

This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

References

1. Baker, M., Aderholdt, F., Venkata, M.G., Shamis, P.: Openshmem-ucx: Evaluation of UCX for implementing openshmem programming model. In: Venkata, M.G., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments - Third Workshop, OpenSHMEM 2016, Baltimore, MD, USA, August 2-4, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10007, pp. 114–130. Springer (2016), http://dx.doi.org/10.1007/978-3-319-50995-2_8
2. Baker, M., Pophale, S., Vasnier, J.C., Jin, H., Hernandez, O.: Hybrid Programming Using OpenSHMEM and OpenACC, pp. 74–89. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-05215-1_6
3. Bonachea, D.: Gasnet specification, v1.1. Tech. rep., Berkeley, CA, USA (2002)
4. ten Bruggencate, M., Roweth, D., Oyanagi, S.: Thread-Safe SHMEM Extensions, pp. 178–185. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-05215-1_13
5. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. pp. 2:1–2:3. PGAS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/2020373.2020375
6. Dinan, J., Balaji, P., Goodell, D., Miller, D., Snir, M., Thakur, R.: Enabling mpi interoperability through flexible communication endpoints. In: EuroMPI 2013. Madrid, Spain (2013)
7. Dinan, J., Flajslik, M.: Contexts: A mechanism for high throughput communication in OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 10:1–10:9. PGAS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2676870.2676872
8. Lawry, W., Wilson, C., Maccabe, A.B., Brightwell, R.: COMB: A portable benchmark suite for assessing MPI overlap. In: 2002 IEEE International Conference on Cluster Computing (CLUSTER 2002), 23-26 September 2002, Chicago, IL, USA. pp. 472–475 (2002), https://doi.org/10.1109/CLUSTR.2002.1137785

9. Namashivayam, N., Knaak, D., Cernohous, B., Radcliffe, N., Pagel, M.: An Evaluation of Thread-Safe and Contexts-Domains Features in Cray SHMEM, pp. 163–180. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-50995-2_11
10. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, pp. 533–546. Springer Berlin Heidelberg, Berlin, Heidelberg (1999), http://dx.doi.org/10.1007/BFb0097937
11. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., Bouteiller, A.: UCX: An open source framework for HPC network APIs and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. pp. 40–43 (Aug 2015)
12. Sridharan, S., Dinan, J., Kalamkar, D.D.: Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints. SC14: International Conference for High Performance Computing, Networking, Storage and Analysis 00, 487–498 (2014)
13. The Ohio State University: OSU micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/ (2016)
14. Venkata, M., Imam, N., Pophale, S., Mintz, T.: OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments: Third Workshop, OpenSHMEM 2016, Baltimore, MD, USA, August 2 – 4, 2016, Revised Selected Papers. Lecture Notes in Computer Science, Springer International Publishing (2016), https://books.google.com/books?id=btS4DQAAQBAJ